FA-LAMP: FPGA-Accelerated Learned Approximate Matrix Profile for Time Series Similarity Prediction

Amin Kalantar
Department of Computer Science and Engineering
University of California Riverside
Email: akala006@ucr.edu

Zachary Zimmerman
Department of Computer Science and Engineering
University of California Riverside
Email: zzimm001@ucr.edu

Philip Brisk
Department of Computer Science and Engineering
University of California Riverside
Email: philip.brisk@ucr.edu

Abstract—**With the proliferation of low-cost sensors and the Internet-of-Things (IoT), the rate of producing data far exceeds the compute and storage capabilities of today's infrastructure. Much of this data takes the form of time series, and in response, there has been increasing interest in the creation of time series archives in the last decade, along with the development and deployment of novel analysis methods to process the data. The general strategy has been to apply a plurality of similarity search mechanisms to various subsets and subsequences of time series data in order to identify repeated patterns and anomalies; however, the computational demands of these approaches renders them incompatible with today's power-constrained embedded CPUs.**

To address this challenge, we present FA-LAMP, an FPGA-accelerated implementation of the Learned Approximate Matrix Profile (LAMP) algorithm, which predicts the correlation between streaming data sampled in real-time and a representative time series dataset used for training. FA-LAMP lends itself as a real-time solution for time series analysis problems such as classification and anomaly detection, among others. FA-LAMP provides a mechanism to integrate accelerated computation as close as possible to IoT sensors, thereby eliminating the need to transmit and store data in the cloud for posterior analysis.

At its core, LAMP and FA-LAMP employ Convolution Neural Networks (CNNs) to perform prediction. This work investigates the challenges and limitations of deploying CNNs on FPGAs when using state-of-the-art commercially-supported frameworks built for this purpose, namely, the Xilinx Deep Learning Processor Unit (DPU) overlay and the Vitis AI development environment. This work exposes several technical limitations of the DPU, while providing a mechanism to overcome these limits by attaching our own hand-optimized IP block accelerators to the DPU overlay. We evaluate FA-LAMP using a low-cost Xilinx Ultra96-V2 FPGA, demonstrating performance and energy improvements of more than an order of magnitude compared to a prototypical LAMP deployment running on a Raspberry Pi 3. Our implementation is publicly available at https://github.com/fccm2021sub/fccm-lamp.

I. Introduction

The proliferation of IoT sensors and the volume of data that they generate creates unique challenges in edge computing [1]. One motivating application, among many, is real-time seismic event prediction, which can inform hazard response strategies and enhance early warning systems presently deployed [2–4]. In this case, the relevant question is whether or not the most recent seismic measurements strongly correlate to the relatively short window of time leading up to a previously observed seismic event. Such a system could benefit by increasing the throughput of the near-sensor raw data processing, and acceleration using an FPGA represents one potential avenue to do so.

This paper describes an FPGA-based accelerator for a streaming time series prediction scheme called the *Learned Approximate Matrix Profile (LAMP)* [5]. Given the most recent window of data points, LAMP uses a *Convolutional Neural Network (CNN)* to predict whether or not a similarly correlated pattern occurred in the time series used to train the model.

While, in theory, these correlations could be computed exactly, the exact methods are impractical due to the requirement that the streaming time series be archived, and the fact that computing them entails execution of an $O(n^2)$ algorithm on a time series of ever-increasing length [6]. It is certainly more practical to perform inference on a moderately sized CNN; nonetheless, the overhead of CNN inference remains a computational bottleneck that limits the achievable sampling rate. Embedded CPU-based solutions are state-of-the-art, but higher performance and lower energy consumption could be achieved through FPGA acceleration. We call our approach *FPGA-Accelerated LAMP*, or *FA-LAMP*, for short.

We implemented and evaluated FA-LAMP on a Xilinx Zynq UltraScale+ MPSoC, compiling it to run on a Xilinx *Deep Processing Unit (DPU)* overlay using the Vitis AI development environment. Several layers of the CNN were not compatible with the DPU; to complete the system, we implemented these layers in software to run on an integrated ARM CPU, and also used high-level synthesis (HLS) tools to generate custom IP block accelerators, which achieved significantly higher throughput. One challenge involved the output layer, which computes a sigmoid activation function; we considered two approximations and evaluated them in terms of accuracy, performance (throughput), resource utilization, and energy consumption on three time series datasets from the domains of seismology, entomology, and poultry farming. Our highest-performing FA-LAMP system configuration achieved throughput of 453.5 G operations per second with an inference rate 10.7× faster and an 15.8× improvement in energy consumption compared to running LAMP on a Raspberry Pi; our most accurate configuration achieved throughput of 428.3 G operations per second with an inference rate 10.1× faster and an 11.6× improvement in energy consumption. Using a dataset obtained from the entomology domain, we show how LAMP can be combined with a post-processing classifier to better understand insect feeding behavior.

The rest of the paper is organized as follows. Section II introduces the LAMP concept and present the FA-LAMP acceleration system, including the opportunities provided and constraints imposed by Xilinx DPU technology. Section III describes the experimental setup and evaluation methodology in detail. Section IV reports the result of our experimental evaluation. Section V summarizes related work that encompasses deep neural network acceleration using FPGAs. Section VI concludes the paper and outlines directions for future work.

II. FA-LAMP System Overview

A. Background: The Matrix Profile

A *Time series* $T = \langle t_1, t_2, \ldots, t_n \rangle$ is an ordered sequence of n scalar data points called a *subsequence*. A *subsequence* of length m and starting at position i is denoted $t_{i,m}$, or t_i if m is known from context. The *Pearson correlation* between subsequences $t_{i,m}$ and $t_{j,m}$, which measures their similarity is denoted $c_{i,j}$; ($c_{i,j}$ values closer to 1 indicate strong similarity; values closer to 0 indicate dissimilarity).

All of the $c_{i,j}$ values could be computed by pairwise enumeration among subsequences and the results could be stored in a matrix; however, the cost of storing the matrix would be prohibitive, especially for large time series. For many time series analysis applications, it suffices to retain the maximum correlation value for each subsequence, rather than all of them. Subsequence t_j is defined to be the *nearest neighbor* of subsequene t_i if $c_{i,j} \geq c_{i,k}, \forall k \neq j$.

The *Matrix Profile (MP)* [6] (see Fig. 1) is a vector that contains the correlations of the nearest neighbors of each subsequence in T: $P(T) = \langle c_i^{max} | \ 1 \leq i \leq n - m + 1 \rangle$, where c_i^{max} is the maximum correlation between t_i and any other subsequence $t_j \in T$, excluding subsequences near t_i.

B. Background: LAMP

The MP is itself a time series and has many uses in time series analysis; while the MP can be computed efficiently with GPUs [6], doing so requires the entire time series and is not amenable to streaming data. An alternative is LAMP [5], which predicts the correlation of the most recent length-m window of streaming data points to a representative time series used to train the model. The objective of this work is to accelerate the LAMP inference procedure on a low-cost FPGA.

Fig. 2 illustrates the LAMP inference process. Each input consists of $\mathbf{J}$ z-normalized subsequences of length $\mathbf{M}$, extracted with stride $\mathbf{S}$. This scheme defines an extraction window in the data, $\mathbf{W}$, where $||\mathbf{W}|| = \mathbf{J} \cdot \mathbf{S} + \mathbf{M} - \mathbf{1}$. We slide $\mathbf{W}$ across the time series and extract a new input for the model for each position of $\mathbf{W}$. This procedure generates vectors of length $\mathbf{M}$ with $\mathbf{J}$ channels as inputs to LAMP's neural network (a CNN), shown in Fig. 3. For each input, the model predicts $\mathbf{J} \cdot \mathbf{S}$ LAMP values, one for each subsequence in $\mathbf{W}$.

LAMP's CNN is a simplified version of ResNet [7] for time series classification [5, 8]. Every convolutional layer in the model is preceded by a batch normalization layer (omitted from Fig. 3 for simplicity). The output layer uses a sigmoid activation function to enable regression. The model inputs and outputs are modified to support multiple predictions at once.

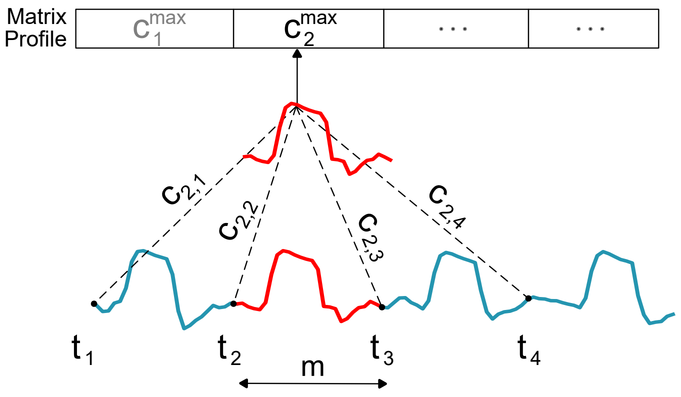

Figure 1. Pearson correlation is computed between the second and the rest of subsequences. The process is applied to all the subsequences in the time series. Matrix Profile is a vector containing the maximum correlation found for each subsequence.

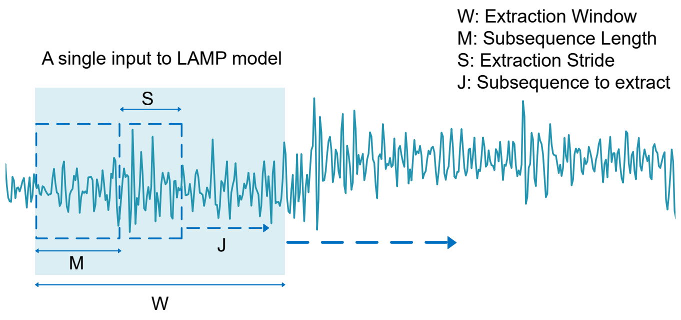

Figure 2. Overview of LAMP inference.

C. Objective

Our plan was to accelerate LAMP neural network inference on a Xilinx Ultra96-V2 FPGA board, leveraging the Xilinx DPU overlay to achieve a balance between performance and programmability. The on-board Xilinx UltraScale FPGA features an integrated dual-core ARM CPU, and has sufficient capacity to realize at most one DPU, with sufficient logic remaining to implement custom IP block accelerators.

We ran into several technical challenges, which we outline below; the following subsections describe our solutions. First, the DPU does not support the Global Average Pooling (GAP) and sigmoid layers, shown on the right-hand-side of Fig. 3; these layers must be implemented in software running on one of the ARM CPU cores or as custom hardware IP block accelerators. Second, implementing the fully connected layer, which sits between the GAP and sigmoid layers, would entail significant data transfer overhead between the DPU and the ARM CPU / IP block. Third, the DPU uses different configurations to perform the convolutional layer (including accumulation and ReLUs); with space for just one DPU, dynamic reconfiguration during inference is needed to support the fully connected layer; the alternative, which we adopted, is to implement the fully connected layer externally as well.

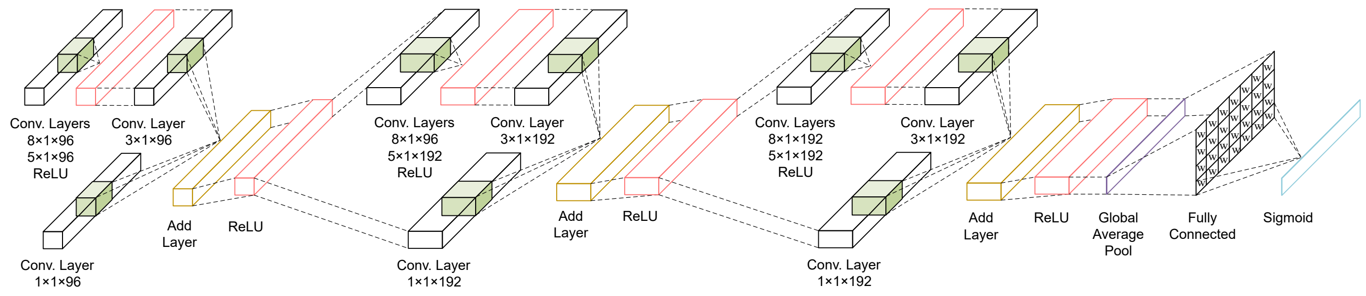

Figure 3. Proposed neural network for LAMP. Every convolutional layer is preceded by a batch normalization layer.

D. DPU Overlay

The Xilinx DPU is a programmable overlay that accelerates many common CNN operations, such as convolution, deconvolution, max pooling, and fully connected layers [9]; Fig. 4 depicts its architecture, which we do not describe in detail due to space limitations. The DPU features user-configurable parameters to optimize resource utilization and to select which features are needed for a given deployment scenario. For example, our implementation does not use softmax, channel augmentation, and depthwise convolution. Seven architectural variants of the DPU exist, with IDs ranging from B512 (smallest) to B4096 (largest); the largest variant that we consider in this study is the B2304.

The DPU compiler translates a neural network model into a sequence of DPU instructions. After start-up, the DPU fetches these instructions from off-chip memory to control the compute engine's operations. The compute engine employs deep pipelining and comprises one or more processing elements (PE), each consisting of multipliers, adders, and accumulators. DSP blocks can be clocked at twice the frequency as general logic.

The DPU buffers input, output, and intermediate values in BRAM to reduce external memory bandwidth. The DPU directly connects to the Processing System (PS) through the Advanced eXtensible Interface 4 (AXI4) to transfer data. The host program uses the Xilinx Deep Neural Network Development Kit (DNNDK) to control the DPU, service interrupts, and coordinate data transfers. In our design, data transfers were necessary as the final three layers of the CNN (GAP, fully-connected, and sigmoid) were performed outside of the DPU.

E. HLS Kernel

This subsection summarizes the steps taken to design an IP accelerator that performs the GAP, fully connected, and sigmoid layers using High-Level Synthesis (HLS). Except when explicitly stated otherwise, the discussion that follows assumes the use of a 32-bit floating-point data format.

(1) Global Average Pool (GAP): The output of the final convolutional layer in Fig. 3 is an array of feature maps $D \in \mathbb{R}^{M\times N}$ corresponding to each of the N channels. The GAP generate an N-dimensional vector $q \in \mathbb{R}^N$ consisting of the

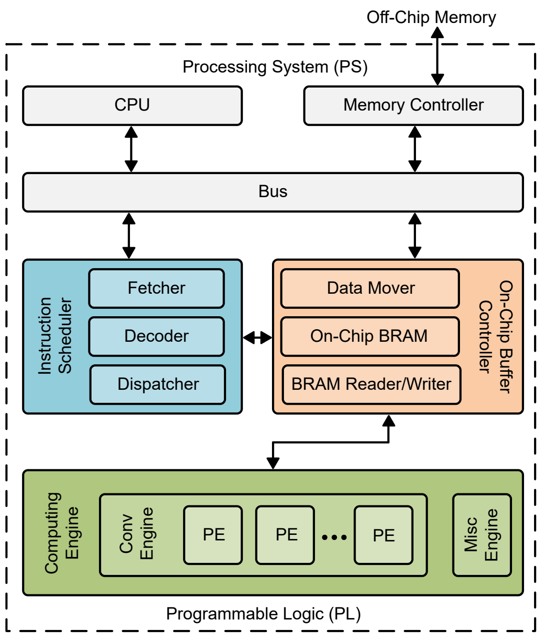

Figure 4. DPU hardware architecture consisting of computing arrays, instruction fetch unit, and a global memory pool module.

average value of each feature map. In other words,

$$q_j \leftarrow \frac{1}{M}\sum_{i=1}^{M} D_{i,j}, \quad 1 \le j \le N. \tag{1}$$

The vector q is then passed to the fully connected layer.

(2) Fully Connected Layer: The input to the fully connected layer is a feature vector $q \in \mathbb{R}^N$. The fully connected layer left-multiplies a weight matrix $W \in \mathbb{R}^{N\times M}$ by q and adds a bias vector $b \in \mathbb{R}^M$, to the result, yielding a new feature vector $z \in \mathbb{R}^M$.

$$z \leftarrow qW + b. \tag{2}$$

We use row-wise vector-matrix multiplication and a tiling strategy [10] to boost throughput. Initially, we set $z \leftarrow b$ in BRAM. We then process each feature $q_i, 1 \le i \le N$ and multiply it by the element in the i^{th} row of the weight matrix, $W_{1,j=1\dots M}$, adding each scalar product term to z_j, i.e., $z_j \leftarrow q_i W_{i,j}$, once again, storing the accumulated sum in BRAM. This scheme allows the execution of the fully connected layer to start as soon as the first element q_1 produced by the GAP

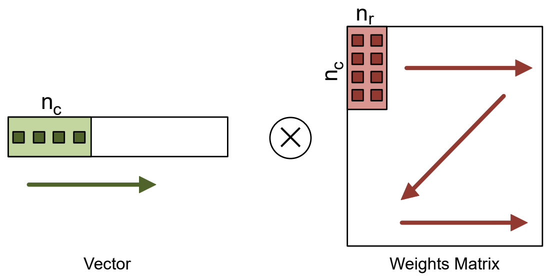

Figure 5. Column-wise vector-matrix multiplication tiling scheme.

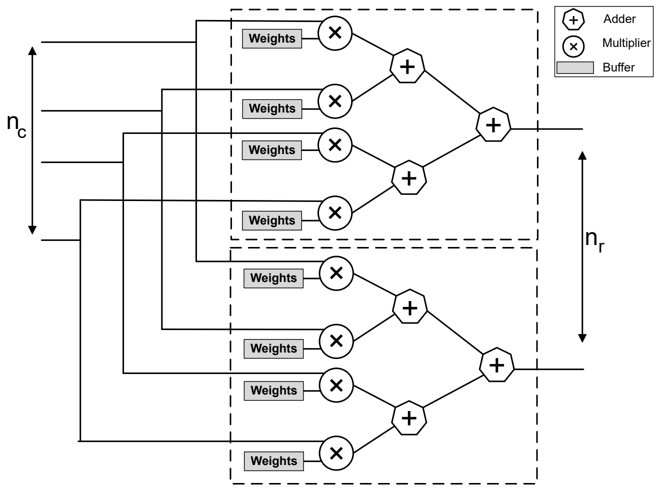

Figure 6. Vector-matrix multiplication unit schematic.

layer arrives; likewise, each feature q_i can be discarded as soon as all of its intermediate products are computed.

To optimize performance, we tile the weight matrix W into small $n_c \times n_r$ blocks as shown in Fig. 5; each vector element is multiplied by n_r matrix elements, allowing the accelerator to perform $n_c \times n_r$ scalar multiplication operations per cycle. Parameter n_c must be chosen to make sure that the latency of GAP layer is greater than the number of cycles required to process n_c vector elements; n_r is chosen to be as large as possible to increase system parallelism, subject to resource constraints. Fig. 6 illustrates the tiling scheme for $n_c = 4$ and $n_r = 2$; we set $n_c = 8$ and $n_r = 4$ in our experiments.

(3) Sigmoid Activation: The LAMP neural network applies the sigmoid activation function to each scalar element of the feature vector z produced by the fully connected layer. To simplify notation, we present the sigmoid function of a scalar input x which can represent any of the scalars $z_i \in z$:

$$f(x) = \frac{1}{1+e^{-x}} \tag{3}$$

Computing the sigmoid function directly on an FPGA is impractical due to the cost of division and exponentiation. Informed by extensive studies regarding sigmoid approximations [11], we chose two variants to evaluate: ultra_fast_sigmoid, a piecewise approximation used in the machine learning framework library Theano [12]; and sigm_fastexp_512, which expands the exponential function for an infinite limit [13].

There are inherent tradeoffs among these approximations in terms of accuracy, throughput/latency, area, and energy consumption; additionally, their implementation differs radically, depending on the chosen precision and whether they are implemented using fixed- or floating-point arithmetic[1]. A thorough survey of the tradeoffs involved is beyond the scope of this paper. The final design, which we evaluate in the following section, uses 8-bit fixed-point arithmetic.

The ultra_fast_sigmoid approximation is defined in terms of $2x$ rather than x to simplify the presentation:

$$f(2x) = \begin{cases} 0.5(\frac{1.5x}{1+x}+1) & 0 \leq x < 1.7 \\ 0.5(1+0.935+ \\ 0.045(x-1.7)) & 1.7 \leq x < 3 \\ 0.5(1+0.995) & x \geq 3 \\ 0.5(-\frac{-1.5x}{1-x}+1) & -1.7 \leq x \leq 0 \\ 0.5(1-(0.935+ \\ 0.045(-x-1.7))) & -3 < x \leq -1.7 \\ 0.5(1-0.995) & x \leq -3 \end{cases} \tag{4}$$

Due to the relative simplicity of the operations compared to directly computing the sigmoid function, ultra_fast_sigmoid can be implemented as a low-latency kernel.

The sigm_fastexp_512 approximation expands the exponential function in terms of an infinite limit ($n \to \infty$), using a value of $n = 512$ to render the approximation computable [13]:

$$exp(x) = \prod_{k=1}^{lg(n)} (1+\frac{x}{k})^k, \quad n = 512 \tag{5}$$

$$sigm(x) = \frac{1}{1+exp(-x)} \tag{6}$$

We implemented our sigmoid layer in HLS using a loop that takes x as an input from the fully connected layer and approximates the sigmoid using either Eq. (4) or Eq. (6). In both scenarios, we pipelined the loop with an Initiation Interval (II) of 1; the latency of the loop for sigm_fastexp_512 is higher due to the complexity of the operations.

Fig. 7 shows the sigm_fastexp_512 and ultra_fast_sigmoid approximations, along with their associated errors, which we computed as the squared difference between them and an exactly-computed sigmoid function. Neither is uniformly more accurate than the other for all reported values of x, but ultra_fast_sigmoid has noticeably higher error closer to zero. This error is shown to be tolerable in classification problems [14], where results is normally determined through comparison, not exact values. The error has a greater impact for regression systems that subsequently process the neural network's calculated output.

(4) HLS Optimizations: We optimized our design using directives provided by Vivado HLS and through manual redesign of the fully connected layer. As shown in Fig. 8,

[1]Alternative implementations, such as logarithmic number systems or Posits, are also possible, but are neither discussed nor evaluated here.

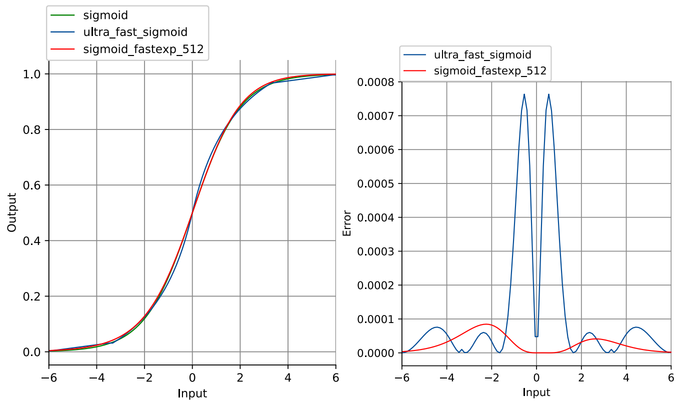

Figure 7. (a) Approximation functions for sigmoid and (b) their error. Both charts were computed using 32-bit floating-point data.

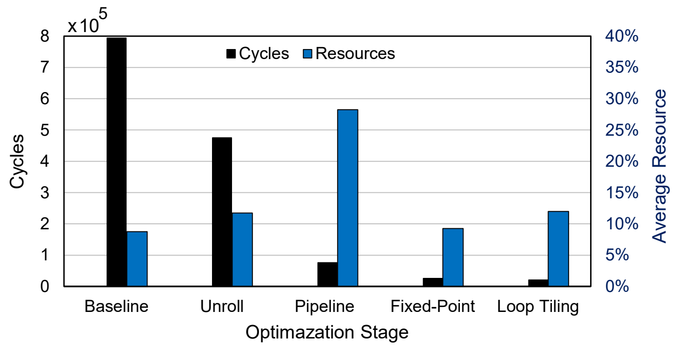

Figure 8. Incremental improvements in the custom kernel latency and resource utilization due to HLS optimizations.

we achieved a 20× speedup over our baseline implementation, while increasing resource usage by 1.5×:

- Baseline : our starting point.
- Unroll : unrolls the inner loops of the GAP and fully connected layers.
- Pipeline : pipelines the outer computation loops and I/O interface loops to infer burst reads/writes; the three layers execute as a pipeline to maximally overlap computation.
- Fixed−Point : is the design implemented in an 8-bit fixed-point UINT data format which reduces the resource utilization by 3× [15].
- Loop−Tiling tiling the fully connected layer (see Fig. 5), while retaining the 8-bit UINT data format.

Most of the speedup arises from pipelining and unrolling loops. The increased resource utilization in Unroll and Pipeline designs is due additional registers and DSP slices.

Fig. 9 shows the overall design. The HLS kernel implements the GAP, fully connected, and sigmoid layers while the rest of the neural network runs on the DPU. The DPU and HLS kernel connect to the processing system via AXI4 ports to allow access to the DDR memory space. The Zynq UltraScale+ processing system in our platform has four High-Performance (HP) ports and two High-Performance Cache coherent (HPC) ports. The DPU I/O interfaces and HLS kernel connect to the HP ports, which provide lower latency than the HPC ports; the DPU instruction fetch port connects to an HPC port.

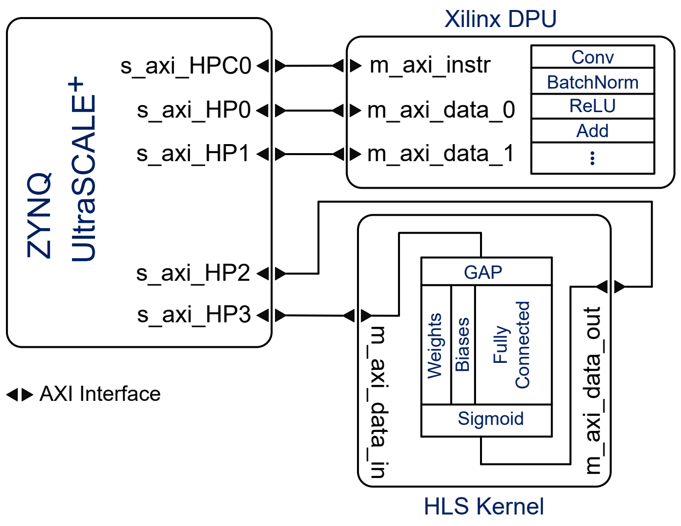

Figure 9. The FA-LAMP implementation comprises a Zynq UltraScale+ processing system, DPU IP, and custom HLS kernel.

III. Experimental Setup

A. Model Training

FA-LAMP deployment on an FPGA begins by training the model. We set the number of subsequences **J** to 32, the length of window **M** to 100, and the stride **S** to 8. We used the Adam [16] optimizer to train the model using stochastic gradient descent with a learning rate of 1e-3 and a batch size of 128. The training objective is to minimize the mean squared error loss between the predicted and exact MP values for the training data set. As a performance optimization, we rearranged the layers in the original LAMP CNN design [5] so that each convolutional layer is followed by a batch normalization layer followed by a ReLU layer; this enables batch normalization to merge with the convolution layer in the DPU.

We train a LAMP model for each dataset offline using an Nvidia Tesla P100 GPU. The trained network is partitioned into two parts: (i) the layers to be executed on a custom kernel (i.e., last three layers: fully connected, GAP, and sigmoid), and (ii) the rest of the model, which maps onto the DPU. We store the weights and activations of the fully connected layer in a header file. The code for the custom kernel later includes this header file for high-level synthesis, noting that the sigmoid and GAP layers do not have any parameters. The second sub-graph of the model is stored in the h5 format file for compilation and deployment on the DPU.

B. Model Inference

We use Vitis AI to quantize and compile the trained LAMP model. AI Quantizer converts all of the model weights and activations into a fixed-point INT8 format. The AI Compiler then maps the model to the DPU instruction set and data flow. We specified the custom kernel (fully connected, GAP, and sigmoid layers) in C++ using the same 8-bit fixed-point data type. As mentioned in Section II, we compared two sigmoid implementations: ultra_fast_sigmoid and sigmoid_fastexp_512.

We synthesized the custom kernel using Vivado HLS 2019.2 and integrated the resulting IP block with the DPU using the Vitis 2019.2 flow for synthesis, placement, and routing.

We evaluated the LAMP CNNs on a Xilinx Ultra96-V2 FPGA board, which integrates two dual-core CPUs (an ARM Cortex A-53 operating at 1.5 GHz and an Cortex-R5 operating at up to 600 MHz) with a Xilinx Zynq UltraScale+ MPSoC featuring 70,560 LUTs, 360 DSP slices and 7.5 MB of BRAM. We used a 16 GB SD card to store an embedded Linux image created with PetaLinux 2019.2 along with the input time series datasets for the design that we will use for inference. We wrote a host program in C++ that uses the DNNDK API to communicate with the DPU IP core.

In the standard DPU flow, unsupported layers are offloaded to the host processor rather than custom IP blocks. We implemented the custom kernel layers on the Cortex-A53 core, because it supports a higher clock frequency than the Cortex-R5 core, and measure its performance against our proposed accelerator. The source code running on the ARM processor also uses the 8-bit fixed-point UINT data type.

We also ported the entire LAMP inference engine to a Raspberry Pi 3 board (32-bit floating-point implementation), which we used for 100% software baseline. We choose Raspberry Pi 3 for our baseline, since Keras [17] does not provide support for the Ultra96-V2 board; thus, it is not presently possible to compile the full LAMP model to either of the two available ARM cores on the board.

C. Measurements

We report the throughput and the energy consumption of FA-LAMP neural network inference by direct execution of the model on the aforementioned mentioned platforms using three time series datasets, which are summarized in the next subsection. The throughput is reported as the number of multiply-accumulation operations in the model (7.71 GOP) executed per second.

We also report the *inference rate* of each platform, which we define to be the number of Matrix Profile values predicted per second. We measure the Ultra96-V2 and Raspberry Pi power consumption using a commercially available Kuman power meter, which provides power measurements for the entire board. We report total energy consumption by multiplying the power measurement by the time it takes to execute inference on a batch of 128. Every batch of data predicts 256 MP values based on the configured LAMP parameters, for a total of 128×256 predictions per inference. Batch sizes larger than 128 led to degraded results on the Raspberry Pi, so we reported results that represent a best-case for our baseline. We report resource utilization results from Vivado's post-implementation reports.

As mentioned in Section II, the user can configure the DPU's degree of parallelism and its parameters for resource utilization. We evaluated the efficiency of all DPU variants that we could fit onto the Ultra96-V2 platform. The number in each DPU variant's name indicates it's peak throughput, e.g., a DPU B512 can perform up to 512 operations per cycle. The Ultra96-v2 board can fit no more than one DPU core. We set the DPU's BRAM and DSP usage to low and disabled the average pool and softmax instructions since the LAMP neural network does not perform these operations. The DPU IP provides two distinct clock inputs:we set the input clock for DSP blocks to 300 MHz and the input clock for general logic to 150 MHz.

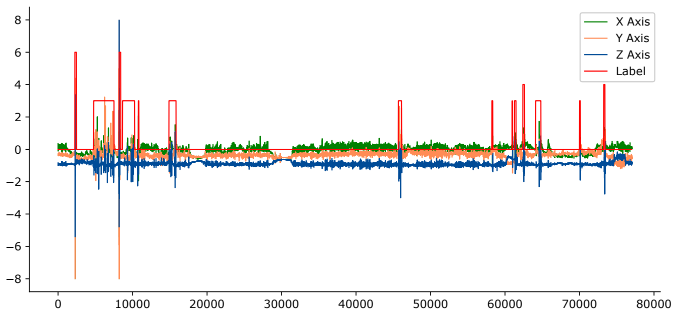

Figure 10. A snippet of chicken accelerometer data with corresponding labels (Preening: label height = 3, dustbathing: label height = 4, and pecking: label height = 6).

D. Benchmarks

We trained neural networks for three time series datasets and measured the error of the model's predictions; this methodology is similar in principle to prior work on LAMP [5].

(1) Seismology Domain: The Earthquake dataset is obtained from a seismic station [6]. Real-time event prediction impacts seismic hazard assessment, response, and early warning systems [2–4]. We split the time series into 120 million data points for training and 30 million data points for inference.

(2) Entomology Domain: The Insect EPG dataset is obtained from an Electrical Penetration Graph (EPG) that records insect behavior [6]. This time series is the record of an insect feeding on a plant and observed behaviors were classified by an entomologist as Xylem Ingestion, Phloem Ingestion, or Phloem Salivation. Understanding feeding behavior of insects can help farmers identify vector-bearing pests that may decimate crops. We split the time series into 2.5 million data points for training and 5 million data points for inference.

(3) Poultry Farming Domain: The Chicken Accelerometer dataset was collected by placing a tracking sensor on the back of a chicken [18]. The sensor outputs acceleration measurements along the x-, y-, and z-axes at a 100 Hz sampling frequency. The data was labeled to classify the chicken's behavior into one of three categories: *Pecking*, *Preening*, or *Dustbathing*. This classification allows farmers to detect and stop the spread of diseases among the poultry. For example, infected chickens exhibit a marked increase in preening and dustbathing behavior compared to uninfected chickens. Fig. 10 depicts a snippet of the dataset corresponding to the x-, y-, and z-axes and behavioral labels. Using only the x-axis measurements, we split the time series into six million data points for training and 2 million data points for inference.

E. Source code and Data Availability

All the data, code, and models used to produce the results in this paper is available and released [19].

Table I
THROUGHPUT (GOPS) AND RESOURCE UTILIZATION COMPARISON BETWEEN DIFFERENT DPU ARCHITECTURES; (DPU + IP) USES A B2304 DPU.

	DPU + ARM (B512)	DPU + ARM (B800)	DPU + ARM (B1024)	DPU + ARM (B1152)	DPU + ARM (B1600)	DPU + ARM (B2304)	DPU + IP (ultra_fast)	DPU + IP (fastexp_512)
Logic Usage	39K (56%)	42K (59%)	46K (65%)	44K (62%)	49K (70%)	52K (75%)	57K (82%)	60K (86%)
Register Usage	50K (36%)	57K (40%)	65K (46%)	64K (45%)	77K (54%)	87K (62%)	95K (67%)	100K (71%)
DSP Usage	78 (21%)	117 (32%)	154 (42%)	164 (45%)	232 (64%)	289 (79%)	290 (80%)	326 (90%)
On-chip RAM Usage	77 (35%)	95 (44%)	109 (50%)	127 (58%)	131 (60%)	171 (79%)	174 (81%)	174 (81%)
Throughput	70.4	107.0	154.2	167.6	220.2	367.1	**453.5**	**428.3**
Peak Throughput	153	240	307	345	480	691	691	691

IV. RESULTS

A. Throughput and Resource Utilization

Table I summarizes the resource utilization and the measured throughput of FA-LAMP inference using various system configurations on the Ultra96-V2 FPGA board. The DPU + ARM columns report results when the custom kernel (fully connected, GAP, and sigmoid layers) run on the ARM CPU, while the DPU + IP columns report results for the custom kernel implemented as FPGA IP blocks that connect directly to the DPU; the largest and best-performing B2304 DPU is used when reporting results for DPU + IP. Results are reported for the custom kernel implemented using two sigmoid approximations: ultra_fast_sigmoid (ultra_fast) and sigmoid_fastexp_512 (fastexp_512) to approximate the sigmoid function.

The DPU + ARM results in Table I show that system throughput increases as DPU size and complexity increases, from B512 to B2304. The highest overall throughput is achieved for the DPU + IP configurations, as the three custom kernel layers that the DPU cannot execute are moved from the ARM CPU to a custom accelerator. Data transfer overhead remains present in both cases between the DPU and ARM CPU / IP block: each read for an input batch of data takes around 0.3 ms and each write takes around 0.1 ms; the port throughput is around 250 MB/s.

Table I also reports the peak (achievable) DPU throughput for each system configuration; this does not include the throughput of the ARM CPU or IP block because the inference procedure, at present, does not lend itself to concurrent execution. The percentage of achievable throughput ranges from 43.6% to 53.1% for the DPU + ARM configurations, and jumps to 65.6% and 62.0% for the two DPU + IP configurations. Even if a hypothetical next-generation DPU could support the three custom kernel operations, the overhead of DPU reconfiguration, which we avoided in the design(s) evaluated here, would also limit the achievable throughput.

DPU resource utilization depends on the degree of parallelism in the chosen configuration; on-chip RAM buffers the weights, bias, and intermediate features. As DPU I/O channel parallelism increases, more on-chip RAM is needed to store more intermediate data and more DSP slices are needed to process that data. When the low DSP usage option is chosen, the DPU uses DSP slices exclusively for multiplication in the convolution layers and offloads accumulation to LUTs. This explains the observed increase in LUT usage as DPU throughput increases.

Table II
FA-LAMP NEURAL NETWORK INFERENCE ACCURACY.

Time Series Dataset		FA-LAMP Inference Accuracy		
Name	Train / Test Split	32-bit float	Proposed: ultra_fast	Proposed: fastexp_512
Earthquake	120M / 30M	97.4%	91.4%	92.5%
Insect EPG	2.5M / 5M	97.2%	90.8%	93.2%
Chicken Accel.	6M / 2M	95.8%	86.9%	91.1%

The custom IP kernels consume additional resources. sigmoid_fastexp_512 performs more multiplication operations and constant division operations than ultra_fast_sigmoid, noting that the latter performs mostly constant multiplications. As a consequence, ultra_fast_sigmoid achieves higher throughput and lower resource utilization compared to sigmoid_fastexp_512; however, as we will see in the next subsection, these benefits come at the cost of lower accuracy.

B. Inference Accuracy

Table II summarizes the accuracy of the FA-LAMP neural network models that we evaluated in the preceding section. We include results for a 32-bit floating-point CPU-only implementation of the FA-LAMP models to quantify the loss in accuracy due to quantization, which is 4.0–4.9% for sigmoid_fastexp_512, and 6.0–8.9% for ultra_fast_sigmoid. The 8.9% accuracy loss for the Chicken Accelerometer dataset for ultra_fast_sigmoid can be attributed to the range of values in the input numbers to the sigmoid kernel. Referring back to Section II, we note that sigmoid layer's input values line in the range [-0.12 1.85], where ultra_fast_sigmoid has the largest error, when inference is performed on this dataset.

C. Comparison to a Raspberry Pi 3

Next we compare the performance and energy consumption of FA-LAMP neural network inference running on the Ultra96-V2 FPGA board to a Raspberry Pi 3, being representative of a purely CPU-based edge computing system. Table III reports the throughput (inference rate) and energy consumption (in Joules) of processing a single batch size of 128 on each platform. The runtime of FA-LAMP neural network inference does not depend

Table III
INFERENCE RATE AND ENERGY CONSUMPTION OF LAMP NEURAL NETWORK INFERENCE ON A RASPBERRY PI 3 AND ULTRA96-V2 BOARD.

	Raspberry Pi 3	DPU + ARM	DPU + IP ultra_fast	DPU + IP fastexp_512
Inf. Rate (Hz)	1.4K	12.1K	15.0K	14.2K
Energy (J)	105.8	7.2	6.7	9.1

on the size of the representative dataset used for training; thus, the inference rate and energy consumption is identical across all datasets.

Both the inference rate and energy consumption of all three Ultra96-V2 FPGAs improve by 1-2 orders of magnitude compared to the Raspberry Pi; according to our power measurements, the Ultra96-V2 FPGA board consumed $\sim$3W of power compared to $\sim$4W for the Raspberry Pi. As expected, the DPU + IP options achieve a higher inference rate than the reported DPU + ARM configuration. Notably, the DPU + IP option using sigmoid_fastexp_512 consumes more energy than both the DPU + ARM and DPU + IP using ultra_fast_sigmoid; referring back to Table I, this occurs due to the higher demand for DSP blocks (36 more than ultra_fast_sigmoid) which are clocked twice as fast as the FPGA general logic.

D. Case Study: Interpreting the FA-LAMP Output

The Matrix Profile can be computed using existing methods in an offline context [6], where LAMP is used to predict it on streaming data [5]. Regardless of how the Matrix Profile is obtained, subsequent post-processing steps are needed to extract actionable information from it.

As a representative example, we explain how FA-LAMP neural network inference can help a scientist to classify the behavior of an insect in real-time. First, we take the training data (2.5M data points, collected over 7 hours) from an insect feeding on a plant. We then create two classes [5]:

Class **A**:Xylem Ingestion/Stylet Passage

Class **B**:Non-Probing

We take a representative dataset from each class ($\mathbf{R_A}$ and $\mathbf{R_B}$) and train two distinct FA-LAMP models, which we respectively denote as $\mathbf{M_A}$ and $\mathbf{M_B}$. Let $\mathbf{S}$ be a subsequence of streaming data. If $\mathbf{M_A(S)} > \mathbf{M_B(S)}$, we predict that behavior **A** is occurring; if $\mathbf{M_A(S)} < \mathbf{M_B(S)}$, we predict that behavior **B** is occurring; otherwise, the prediction is inconclusive.

For evaluation data we consider the inference data (2.5M data points, collected over the next 5 hours from the same insect), whose behavior has also been labeled by an entomologist to provide ground truth. We observed 98.2% accuracy in the results of classification using FA-LAMP. Fig. 11 shows the time series and the actual and predicted labels reported by the FA-LAMP model for a snippet of test data.

V. RELATED WORK

Most FPGA-based deep neural network studies focus on accelerator design, compilation frameworks, and/or domain-specific overlays. Our work borrows ideas from all three areas. Our HLS-generated IP blocks employ optimizations

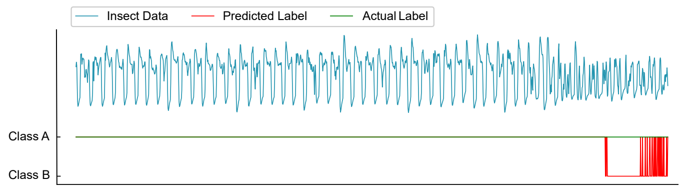

Figure 11. A snippet of insect EPG time series dataset along with the actual and predicted behavior (Class A: label height=1; Class B: label height=0).

such as loop tiling [20], data reuse [21–23], weight-stationary multiplication [24], memory packing [25], fixed-point numeric formats [26], and model weight quantization [27–32].

Commercial frameworks, such as Xilinx's Vitis AI framework, which we used, or Intel's Deep Learning Acceleration (DLA) suite [33], take inspiration from general-purpose languages and frameworks such as HeteroCL [34, 35], as well as frameworks specific to neural networks [36–49].

We used the Xilinx programmable DPU overlay [9], which was optimized for well-known convolutional neural networks [7, 50, 51]; similar overlays include Microsoft's Project Brainwave [52], Intel DLA [53], and Light-OPU [54]. Our work includes mixed use of a programmable overlay and custom IP blocks designed using HLS, and a fairly detailed analysis of how to implement the sigmoid activation function.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an energy-efficient accelerator for time series similarity prediction using CNNs. We integrated a custom IP accelerator block with Xilinx DPU to enable whole-model acceleration of the FA-LAMP CNN on Xilinx Ultra96-V2 FPGA. We analyzed two approximation functions for the output layer of the model in terms of accuracy, performance, resource utilization, and energy consumption, and we showed how FA-LAMP combined with a post-processing classifier can solve a real-world problem. Compared to a Raspberry Pi 3, our design achieved 10.7$\times$ higher inference rate and improved the energy efficiency by 15.8$\times$

We envision several avenues of future work to improve FA-LAMP. We would like to further improve the accuracy of the FA-LAMP CNN by re-training the model after quantization. We also hope to port the FA-LAMP system to a larger and more powerful development board, which would enable the integration of multiple DPUs, along with our custom IP accelerators. We would also like to more thoroughly explore the space of sigmoid approximation functions, including piecewise alternatives to ultra_fast_sigmoid, which might be able to reduce its error, and variants of sigmoid_fastexp_N for values of N other than 512; there is also considerable opportunity to explore the internal architecture and precision of sigmoid_fastexp_N. Long-term, it may be possible to harden the FA-LAMP inference engine so that it can be integrated in the same System-on-a-Chip (SoC) as the sensor(s) whose data it will process.

ACKNOWLEDGMENT

This work was supported in part by NSF Awards #1528181 and #1763795.

References

[1] E. Oyekanlu, "Predictive edge computing for time series of industrial IoT and large scale critical infrastructure based on open-source software analytic of big data," in *IEEE International Conference on Big Data (Big Data)*, 2017, pp. 1663–1669.

[2] S. E. Minson, M.-A. Meier, A. S. Baltay, T. C. Hanks, and E. S. Cochran, "The limits of earthquake early warning: Timeliness of ground motion estimates," *Science Advances*, vol. 4, no. 3, 2018.

[3] R. Allen, H. Brown, M. Hellweg, O. Khainovski, P. Lombard, and D. Neuhauser, "Real-time earthquake detection and hazard assessment by ElarmS across California," *Geophysical Research Letters - GEOPHYS RES LETT*, vol. 36, 2009.

[4] C. Satriano, A. Lomax, and A. Zollo, "Real-time evolutionary earthquake location for seismic early warning," *Bulletin of the Seismological Society of America*, vol. 98, pp. 1482–1494, 2008.

[5] Z. Zimmerman *et al.*, "Matrix profile XVIII: time series mining in the face of fast moving streams using a learned approximate matrix profile," in *IEEE International Conference on Data Mining (ICDM)*, 2019, pp. 936–945.

[6] Z. Zimmerman, K. Kamgar, N. S. Senobari, B. Crites, G. Funning, P. Brisk, and E. Keogh, "Matrix Profile XIV: Scaling time series motif discovery with GPUs to break a quintillion pairwise comparisons a day and beyond," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 2019, p. 74–86.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[8] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," *arXiv preprint arXiv:1611.06455*, 2016.

[9] "DPU for convolutional neural network v3.0, DPU IP product guide." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_0/pg338-dpu.pdf

[10] Z. Que *et al.*, "Optimizing reconfigurable recurrent neural networks," in *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 10–18.

[11] Y. Gao, F. Luan, J. Pan, X. Li, and Y. He, "FPGA-based implementation of stochastic configuration networks for regression prediction," *Sensors*, vol. 20, p. 4191, 2020.

[12] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-farley, and Y. Bengio, "Theano: A CPU and GPU math compiler in python," in *Proceedings of the 9th Python in Science Conference*, 2010, pp. 3–10.

[13] N. G. Timmons and A. Rice, "Approximating activation functions," *arXiv preprint arXiv:2001.06370*, 2020.

[14] S. Decherchi, P. Gastaldo, A. Leoncini, and R. Zunino, "Efficient digital implementation of extreme learning machines for classification," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 59, no. 8, pp. 496–500, 2012.

[15] "Deep learning with INT8 optimization on Xilinx devices," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf

[16] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *3rd International Conference on Learning Representations, ICLR*, 2014.

[17] D. Falbel, "Keras," https://github.com/keras-team/keras.

[18] A. Abdoli, S. Alaee, S. Imani, A. Murillo, A. Gerry, L. Hickle, and E. Keogh, "Fitbit for chickens? time series data mining can increase the productivity of poultry farms," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. ACM, 2020, p. 3328–3336.

[19] FA-LAMP source code repository: https://github.com/fccm2021sub/fccm-lamp.

[20] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2015, p. 161–170.

[21] M. Hardieck, M. Kumm, K. Möller, and P. Zipf, "Reconfigurable convolutional kernels for neural networks on FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2019, p. 43–52.

[22] X. Han, D. Zhou, S. Wang, and S. Kimura, "CNN-MERP: An FPGA-based memory-efficient reconfigurable processor for forward and backward propagation of convolutional neural networks," in *IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 320–327.

[23] Y. Chen, J. Emer, and V. Sze, "Using dataflow to optimize energy efficiency of deep neural network accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 12–21, 2017.

[24] E. Wu, X. Zhang, D. Berman, I. Cho, and J. Thendean, "Compute-efficient neural-network acceleration," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2019, p. 191–200.

[25] L. Petrica, T. Alonso, M. Kroes, N. Fraser, S. Cotofana, and M. Blott, "Memory-efficient dataflow inference for deep CNNs on FPGA," *arXiv preprint arXiv:2011.07317*, 2020.

[26] P. Colangelo, N. Nasiri, E. Nurvitadhi, A. Mishra, M. Margala, and K. Nealis, "Exploration of low numeric precision deep learning inference using Intel FPGAs," in *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 73–80.

[27] Y. Yang, Q. Huang, B. Wu, T. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. Vissers, J. Wawrzynek, and K. Keutzer, "Synetgy: Algorithm-hardware co-design for ConvNet accelerators on embedded FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2019, p. 23–32.

[28] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2016, p. 26–35.

[29] H. Nakahara, Z. Que, and W. Luk, "High-throughput convolutional neural network on an FPGA by customized JPEG compression," in *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 1–9.

[30] K. Ando *et al.*, "Dither NN: An accurate neural network with dithering for low bit-precision hardware," in *International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 6–13.

[31] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. W. Leong, "PIR-DSP: An FPGA DSP block architecture for multi-precision deep neural networks," in *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 35–44.

[32] Z. Xu, J. Yu, C. Yu, H. Shen, Y. Wang, and H. Yang, "CNN-based feature-point extraction for real-time visual SLAM on embedded FPGA," in *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 33–37.

[33] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL™ deep learning accelerator on Arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2017, p. 55–64.

[34] Y.-H. Lai *et al.*, "Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2019, p. 242–251.

[35] K. Kamalakkannan, G. R. Mudalige, I. Z. Reguly, and S. A. Fahmy, "High-level FPGA accelerator design for structured-mesh-based explicit numerical solvers," *arXiv preprint arXiv:2101.01177*, 2021.

[36] H. Sharma *et al.*, "From high-level deep neural models to FPGAs," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[37] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *IEEE/ACM International Conference on Computer-Aided Design, (ICCAD)*, 2018.

[38] C. Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.

[39] H. Zeng, R. Chen, C. Zhang, and V. K. Prasanna, "A framework for generating high throughput CNN implementations on FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2018, pp. 117–126.

[40] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: Automatic generation of FPGA-based learning accelerators for the Neural Network family," in *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[41] J. Xu, Z. Liu, J. Jiang, Y. Dou, and S. Li, "CaFPGA: An automatic gen-

eration model for CNN accelerator," *Microprocessors and Microsystems*, vol. 60, 2018.

[42] P. Xu *et al.*, "AutoDNNchip: An automated DNN chip predictor and builder for both FPGAs and ASICs," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2020, p. 40–50.

[43] R. Zhao, H.-C. Ng, W. Luk, and X. Niu, "Towards efficient convolutional neural network for domain-specific applications on FPGA," in *28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 147–1477.

[44] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," in *27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8.

[45] S. I. Venieris and C. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 40–47.

[46] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2019, p. 73–82.

[47] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," *arXiv preprint arXiv:1802.04799*, 2018.

[48] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Rethinking inference in FPGA soft logic," in *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 26–34.

[49] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 152–159.

[50] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2015.

[51] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[52] J. Fowers *et al.*, "A configurable cloud-scale DNN processor for real-time AI," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 1–14.

[53] M. S. Abdelfattah *et al.*, "DLA: Compiler and FPGA overlay for neural network inference acceleration," *arXiv preprint arXiv:1807.06434*, 2018.

[54] Y. Yu, T. Zhao, K. Wang, and L. He, "Light-OPU: An FPGA-based overlay processor for lightweight convolutional neural networks," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2020, p. 122–132.